APPLICATION FOR UNITED STATES LETTERS PATENT

Software Architecture for Distributed Enterprise Business Applications

Walter Hurst

Kamal Shah

Cross-Reference to Related Application

[0001] This application hereby claims the benefit under Title 35, U.S.C. §119(e)(1) of United States Provisional Patent Application No. 60/421,495, entitled Wakesoft Manager Adapter Distributed Business Application Architecture, filed October 25, 2002, the disclosure of which is incorporated herein by reference in its entirety.

Field of the Invention

[0002] The invention relates to data processing by a digital computer, and more particularly to software architectures for computer applications built on platforms such as Java 2 Enterprise Edition and Microsoft® .net.

Background of the Invention

[0003] Object-oriented programming languages such as Java 2 Enterprise Edition (J2EE) and Microsoft® .net provide businesses with powerful technical infrastructures they can use to build distributed enterprise business applications. Unfortunately, the lack of predefined organization within these technical infrastructures causes applications written on these platforms to be unnecessarily complex and inconsistent. This leads to a variety of problems in developing and building applications that are not addressed by existing tools and technologies.

[0004] One of the problems with known technical infrastructures for distributed enterprise business applications is that they allow unstructured programs to be written. Programmers who code business applications will often write a program

that accomplishes the specific tasks assigned to them without adhering to any predefined, organized structure. As such, when multiple programmers work together to develop a distributed enterprise business application, code written by one programmer may not be easily compatible with code written by another programmer. Furthermore, different programmers may address the same technical issues in different ways. All of this presents problems and creates inconsistencies when the code from the different programmers is combined into one application. This unstructured code also provides for very little reuse, can be hard to maintain (especially if any of the original programmers are no longer available), provides no visibility into how the business requirements are implemented, and tends to solve the same problem in multiple ways which is not productive.

[0005] Another problem with known technical infrastructures for business applications is a lack of separation of concerns. Ideally, programmers should only work on the portions of a business application that they are most proficient at. For example, a programmer proficient at coding business logic should only work on business specific portions of the application and not technical portions. With known technical infrastructures, however, different programmers implementing and coding different portions of a business application are all exposed to the complexity of the technical infrastructure. This requires all of the programmers to have a sophisticated knowledge of the technical infrastructure being used to develop the business application. So programmers who are proficient at implementing business logic must be aware of and deal with technical portions of the business application as well. If these programmers are not proficient at the technical issues, they must do their best to address them. This puts the wrong people in charge of making technical decisions, thus leading to poor quality in the business application. This drawback of conventional distributed enterprise business applications makes it hard to align the right people with the right tasks, and impacts productivity.

[0006] Yet another problem with known technical infrastructures for business applications is that the lack of organization creates inflexible solutions. The lack

of technical flexibility means applications cannot be easily changed when new requirements emerge or when problems are encountered. The lack of technical flexibility also increases costs associated with any technical change.

[0007] Accordingly, there is a need for methods or software that addresses the above mentioned problems to assist programmers in creating improved distributed enterprise business applications on technical infrastructures implementing object-oriented methods such as J2EE and Microsoft® .net.

Summary of the Invention

[0008] The invention is a software architecture that provides a bridge between a software application and its underlying technical infrastructure. The software architecture organizes the application into a set of reusable architectural services implemented with manager objects and adapter objects, while the functionality specific to the software application is organized into discrete application components. The manager objects are instances of provided manager classes and the adapter objects are instances of provided adapter classes. When a software application calls into the software architecture through a specific architecture service to request functionality, a manager object for that service is instantiated to receive the call. The manager object then instantiates the appropriate adapter object and delegates the call to that adapter object. The adapter object instantiates the appropriate application components to provide the functionality requested in the call.

[0009] The software architecture of the invention is composed of a plurality of architecture services, including a navigation service, a business process service, a persistence service, logging service, application state management service, data marshalling service, and a key management service. Each service has its own manager class and at least one adapter class. A service can communicate with another service by having its adapter objects or application components call the manager object of that other service. The use of a plurality of services allows the software architecture to separate different concerns of the software application, for instance, navigation functionality is separated from

business specific functionality and persistence functionality. This enables application functionality for each service to be developed independent of the other services.

Description of the Drawings

[0010] Figure 1A illustrates a conventional business application built directly on a technical infrastructure.

[0011] Figure 1B illustrates a software architecture that provides a bridge between a business application and the underlying technical infrastructure.

[0012] Figure 2 is an abstract illustration of the software architecture of the invention.

[0013] Figure 3 is a UML figure of the software architecture of the invention.

[0014] Figure 4 illustrates code for a manager configuration file.

[0015] Figure 5 illustrates code for an adapter configuration file.

[0016] Figure 6 illustrates a software architecture that includes a plurality of architecture services.

[0017] Figure 7 is a UML figure of a navigation architecture service.

[0018] Figure 8 is a UML figure of a business process architecture service.

[0019] Figure 9 is a UML figure of a persistence architecture service.

[0020] Figure 10 is a UML figure of a logging architecture service.

[0021] Figure 11 is a UML figure of an application state management architecture service.

[0022] Figure 12 is a UML figure of a data marshaling architecture service.

[0023] Figure 13 is a UML figure of a key management architecture service.

[0024] Figure 14 is a flowchart describing how a business application framework retrieves business functionality from the software architecture of the invention.

[0025] Figure 15 is a flowchart describing the operation of the navigation service.

[0026] Figure 16 is a flowchart describing the operation of the business process service.

[0027] Figure 17 is a flowchart describing the operation of the persistence service.

Detailed Description

[0028] The invention is a software architecture that provides structure for software applications, separates different concerns of the application, and provides flexibility that enables changes to be made to software applications with ease relative to conventional applications. As used herein, the term "conventional application" refers to a software application written without the software architecture of the invention.

[0029] The software architecture of the invention defines a structure that can be used for many types of software applications, including but not limited to business applications and distributed enterprise business applications. For clarity, business applications are used herein to describe implementations of the invention. It should be noted, however, that the invention is not limited to business applications.

[0030] The structure provided by the software architecture of the invention can be visualized as a skeletal support that is used as a basis for building a software application. The structure defines intuitive and holistic software components that fill in this skeletal support. These software components are made to be independent so each software component can be replaced or amended without impacting other software components. This makes debugging and modifying a software application an easier task for users.

[0031] The software architecture of the invention separates different concerns of the software application. For instance, users who implement the software architecture of the invention to build applications tend to fall into two categories: technical programmers and application programmers. Accordingly, the software architecture of the invention generally separates technical code

(i.e., technical components) required by the technical programmers from application-specific code (i.e., application-specific components) required by the application programmers. Since these software components are generally made to be independent, changes to one group of software components tend to not impact the other group of software components.

[0032] The software architecture of the invention also provides a simplified application programming interface that exposes functionality necessary to an application programmer but encapsulates the complexity of the technical infrastructure. This technical infrastructure can still be exposed to the technical programmer, or other technical users, outside of this application programming interface to modify and generate code to handle technical issues.

[0033] Additionally, the software architecture of the invention is flexible and configurable to enable a user to implement changes to a software application using methods that are relatively easier than methods used for modifying conventional applications. In an implementation of the invention, the software architecture utilizes adjustable configuration files to provide functionality that would normally be encoded directly into the software application. These configuration files can be easily modified as needed by a user to modify the behavior of the software application. The use of a configuration file therefore enable the re-use of software components and eliminate "hard-coding" of programming behavior in volatile areas that are likely to change through the life of a software application.

[0034] As shown in Figure 1A, a conventional business application 10 is built directly on a technical infrastructure 20. The business application 10 is a computer program developed to carry out business-specific functionality. This functionality is implemented using methods developed and coded by a user. For example, a user familiar with the business requirements of an enterprise can write (i.e., code) and maintain business-specific logic for a business application that the enterprise desires. The underlying technical infrastructure 20 is the programming language platform used for the business application 10. The technical infrastructure 20 is generally implemented on a computer or an

application server. In one implementation, the underlying technical infrastructure 20 is implemented as a J2EE-compliant or a Microsoft® .net compliant application server. Building the business application 10 directly on the technical infrastructure 20 results in the business application 10 having technical code mixed in with business-specific code, leading to functional intermingling that becomes harder to debug and maintain as the business application 10 grows and matures, and as the business-specific logic changes.

[0035] Figure 1B illustrates an implementation of a software architecture 100 constructed in accordance with the invention that provides a bridge between the business application 10 and the underlying technical infrastructure 20. The software architecture 100 provides a middleware layer between the business application 10 and the technical infrastructure 20 that contains structural code generally separating the business application concerns from the technical infrastructure concerns, and generally separating business software components from technical software components. The software architecture 100 provides a framework enabling the business application 10 to interact with the technical infrastructure 20 built on a platform such J2EE or Microsoft® .net.

[0036] Figure 2 illustrates an implementation of the invention where the software architecture 100 separates the technical, structural, and architectural functionality of a distributed application from its application-specific functionality. The technical, structural, and architectural functionality is common to all distributed applications and the software architecture 100 implements this functionality by providing reusable manager objects 210 and adapter objects 212, which are described in more detail below. The application-specific functionality, which is specific to a particular application, is provided by the software architecture 100 using one or more application components 202.

[0037] The application components 202 are discrete portions of code that contain focused functionality specific to the application. For example, if the application is a business application, the application components 202 provide the actual business functionality. An individual application component 202 generally focuses on only a particular piece of this functionality. Many

application components 202 have no knowledge of the technical environment in which they are running. This allows those application components 202 to be greatly independent of their technical environment, enabling them to be reused in other parts of the business application or in completely different software applications. It should be noted that the application components 202 are generated by users of the software architecture 100; they are not provided by the software architecture 100.

[0038] In one implementation of the invention, a business application client 200 calls into the software architecture 100 to request some business functionality. The client 200 is any piece of software that can call into the software architecture 100 to make a request. The software architecture 100 uses the manager object 210 and at least one adapter object 212 to handle this incoming call. The manager object 210 receives and parses the incoming call to determine what is being requested by the client 200. The manager object 210 then selects an adapter object 212 that is configured to handle such a request and delegates the call to the selected adapter object 212. Each adapter object 212 is generally equipped to handle a plurality of tasks. The adapter object 212, upon receiving the call, determines what functionality is requested and calls the appropriate application components 202 to process the request. The use of manager objects 210 and adapter objects 212 to handle calls from the client 200 and to call application components 202 differs from conventional applications where the application code is in control and calls into libraries to accomplish certain specialized functions.

[0039] Figure 3 is a unified modeling language (UML) diagram of the software architecture 100. In the implementation shown in Figure 3, the technical infrastructure 20 is an object-oriented platform (e.g., J2EE), and the software architecture 100 provides at least one manager class 300. The manager class 300 is used by the software architecture 100 to instantiate a manager object 210. The manager class 300 defines all of the properties of the manager object 210, and can define functional methods relevant to incoming calls from the client 200. For instance, the manager class 300 can define a "save" method that is

relevant to an incoming call requesting that a data object be stored in a database. As one of ordinary skill in the art will recognize, in object-oriented programming, a class is a template definition of the properties and behaviors, such as variables and methods, in a particular kind of object. Classes are used to instantiate (i.e., create) objects in memory. Thus, an object is a specific instance of a class; it contains real values instead of variables. A class can have subclasses that can inherit all or some of the characteristics of the class. In relation to each subclass, the class becomes the superclass. Subclasses can also define their own methods and variables that are not part of their superclass. The structure of a class and its subclasses is called the class hierarchy.

[0040] In an implementation of the invention, the software architecture 100 provides a manager configuration 214 that contains configuration data for the manager object 210. When a manager object 210 is instantiated, the manager object 210 retrieves configuration data from its corresponding manager configuration 214. The data in the manager configuration 214 provides the manager object 210 with methods, rules, and filters to apply to incoming calls from the client 200. This includes information specifying which adapter objects 212 to call for different requests from the client 200. A user can modify the behavior of a manager object 210 by modifying the manager configuration 214. This reduces the need to rewrite code for a manager object 210 or manager class simply to modify the behavior of the manager object 210. The manager object 210 can respond dynamically to changes in the configuration data included in the manager configuration file 214. Figure 4 illustrates an implementation of a manager configuration file 214, written in J2EE code, for a manager object named "PersistenceManager."

[0041] Returning to Figure 3, in an implementation of the invention, the software architecture 100 also provides at least one adapter class 302. The adapter class 302 is used by the software architecture 100 to instantiate one or more adapter objects 212 and defines all of the properties and behaviors of the adapter objects 212. As is described below, the software architecture 100 can provide a separate adapter class 302 for each possible type of adapter object 212.

[0042] The adapter objects 212 are pluggable components of the software architecture 100. Since the adapter objects 212 are responsible for calling application components 202 to process requests, the adapter objects 212 provide a mechanism for a user, such as a technical programmer, to modify the behavior of the software architecture 100. This enables the interaction with the technical infrastructure 20 to be changed without modifying the manager objects 210, the application components 202, or the client 200, but by simply modifying the configuration to specify a different adapter object 212. The architecture 100 provides adapter objects 212 for various uses, but also allows technical programmers the option of building their own custom adapter object. This in turn provides a great amount of technical flexibility.

[0043] In an implementation of the invention, the software architecture 100 provides an adapter configuration 216 containing configuration data for the adapter objects 212. As is described below, in one implementation there is a separate adapter configuration 216 for each possible adapter object 212. When an adapter object 212 is instantiated, it retrieves configuration data from the adapter configuration 216.

[0044] The data in the adapter configuration 216 provides the adapter object 212 with methods, rules, and filters to apply to incoming calls from the manager object 210. This includes information specifying which application components 202 to call for different requests from the client 200. A user can modify the behavior of the adapter object 212 by simply modifying the adapter configuration 216. The adapter object 212 can respond dynamically to changes in the configuration data. Figure 5 illustrates an implementation of an adapter configuration 216 for an adapter named "BusinessProcessAdapter" written in Java code.

[0045] Returning again to Figure 3, the software architecture 100 can provide one or more interfaces to be used in conjunction with the manager object 210 and the adapter objects 212. An interface provides templates of behavior that objects can implement. In one implementation, the software architecture 100 provides an adapter interface 304 that enables communications between the

manager object 210 and the adapter object 212 to occur. More specifically, when the manager object 210 instances the adapter object 212, the adapter interface 304 provides methods to be implemented by the adapter object 212 so the manager object 210 can communicate with the adapter object 212.

[0046] As is described below, different manager objects 210 can be associated with different adapter interfaces 304. An adapter object 212 implemented for a specific manager object 210 must therefore implement that manager object's corresponding adapter interface 304.

[0047] The software architecture 100 can also provide one or more interfaces to be used in conjunction with the adapter objects 212 and the application components 202. Similar to the adapter interface 304, in one implementation, the software architecture 100 provides an application component interface 306 that enables communications between the adapter object 212 and the application components 202 to occur. The methods of the application component interface 306 must be implemented by the application components 202. An adapter object 212 can then instance an application component 202 and communicate with that application component 202 after the methods of the application component interface 306 have been implemented.

[0048] Figure 6 illustrates an implementation of the invention where the software architecture 100 organizes its functionality into a plurality of architecture services. In this implementation, the software architecture 100 includes a navigation service 204, a business process service 206, and a persistence service 208. Each architecture service includes one manager object 210 and at least one adapter object 212 to provide technical, structural, and architectural functionality. Accordingly, the software architecture 100 provides a navigation manager class, a business process manager class, and a persistence manager class. Each architecture service also includes a separate manager configuration 214 for its manager object 210. In other implementations, additional architecture services can be included in the software architecture 100 to further organize or increase the functionality of the architecture.

[0049] The navigation service 204 can handle data communications with an end-user of the business application, for instance, through a graphical user interface (GUI). The navigation service 204 can display the GUI to an end-user and can also control the user interface flow or navigation. The business process service 206 can handle the processing of business logic contained in the business application. For example, if the business application is designed to handle banking transactions, the business process service 206 can carry out the associated banking transaction methods, such as transferring funds between accounts. Finally, the persistence service 208 can handle data persistence, such as creating, storing, retrieving, modifying, and deleting objects and data.

[0050] In one implementation, the client 200 initiates a request by calling into the navigation service 204 of the software architecture. The navigation service 204 can route the call to the appropriate service, such as the business process service 206 or the persistence service 208. In another implementation, the client 200 can call directly into the appropriate service it needs (e.g., the business process service 206). End-user interactions, however, are generally initiated at the navigation service 204.

[0051] In an implementation of the invention, each application component 202 is associated with at least one of the services. The software architecture 100 takes application components 202 containing navigation functionality and associates them with the navigation service 204. Likewise, the software architecture 100 associates business logic application components 202 with the business process service 206, and persistence application components 202 with the persistence service 208.

[0052] In one implementation, each service provided by the software architecture 100 can include one or more application component interfaces 306 to be implemented by the application components 202 associated with that service. In an implementation of the invention, each adapter object 212 is associated with a specific application component interface 306, and an application component 202 instanced and called by that adapter object 212 can implement that specific application component interface 306.

[0053] Users, such as the business programmers described above, generate the application components 202 using protocols imposed by the software architecture 100 and the relevant architecture service. In one implementation, one such protocol is that application components 202 must implement the application component interface 306 defined by the relevant architecture service. The service can then call the application components 202 through that application component Interface 306. In an implementation, another such protocol is that the application components 202 must be added to the configuration 216 for an architectural service in a specified format so that they can be read and loaded. The application components 202 of one service can be developed independent from the application components 202 of another service. As such, different users can develop different application components 202 independently of each other.

[0054] The organization of the application components 202 into different services allows users developing portions of a business application to work within a specific service without having to address issues from other services. So a user proficient at developing business logic can generate business logic application components 202 without having to address navigation or persistence issues. Similarly, a user experienced at developing graphical user interfaces can focus on generating navigation application components 202, while a technical user familiar with the technical infrastructure can develop the persistence application components 202. The software architecture 100 then assembles these different application components 202 to form one coherent business application.

[0055] The manager object 210 is the primary point of interaction for the architecture service. Since only one manager object 210 is defined for each architecture service, a relatively small number of manager objects 210 exist, thereby reducing the complexity of the software architecture 100 from the perspective of the client 200. The client 200 can then access the functionality contained in the software architecture 100 by simply communicating with the small number of manager objects 210.

[0056] When the functionality of an architecture service is needed by a user coding a client 200, the user need only write code that calls the manager object 210 associated with the required service. The user does not have to write code that includes the functionality of the service. Thus, the manager object 210 provides an abstraction of the service for the user which simplifies the development of a business application. Decision-making processes are handled by the manager object 210 once it is called, thereby removing that task from the user. The manager object 210 calls an adapter object 212 that encapsulates the actual behavior. The manager object 210 communicates with the adapter object 212 through the adapter interface 304 defined for the relevant architectural service.

[0057] Figure 7 is a UML diagram of one implementation of the navigation architecture service 204 provided by the software architecture 100. For this implementation (as well as the implementations described in Figures 8 through 13), the technical architecture 20 is a J2EE application server. In other implementations, however, other technical architectures can be used. It should be noted that the names used for the objects discussed in Figures 7 through 13 below are merely for explanatory purposes and should in no way be construed as imposing limitations on the invention.

[0058] In the implementation of Figure 7, the software architecture 100 provides a root class 700 called BaseFrameworkObject 700, which is based on the java.lang.Object class provided in the J2EE technical infrastructure. The root class 700 can add at least one method to the java.lang.Object class, for instance, a method to return the class name for the instance of this object. The root class 700 can serve as the root object for all objects used by the software architecture 100, which facilitates adding behavior that will be common to all of them. This also allows all of the software architecture 100 objects to be referenced in a uniform fashion that can be more useful than java.lang.Object.

[0059] The software architecture 100 provides a manager class 300, called BaseManager class 300, which extends the root class 700. The BaseManager class 300 functions as a superclass for all of the manager classes and facilitates

adding behavior that will be common to just the manager objects 210. Using the BaseManager class 300, the software architecture 100 instantiates a manager object 210 called NavigationManager object 210. The NavigationManager object 210 extends the BaseManager class 300 and functions as the manager object 210 for the navigation service 204. The NavigationManager object 210 can implement one or more methods to process incoming calls, select adapter objects 212, and delegate calls to the selected adapter objects 212. The NavigationManager object 210 loads configuration data from the manager configuration 214 (referred to here as NavigationManager configuration 214).

[0060] The software architecture 100 provides an adapter class 302, called BaseAdapter class 302, which is also based on the root class 700 (i.e., BaseFrameworkObject) provided by the software architecture 100. The BaseAdapter class 302 extends the root class 700 and functions as a superclass for all of the software architecture 100 adapter classes. The BaseAdapter class 302 facilitates adding behavior that will be common to just the adapter objects 212, regardless of the service in which the adapter object 212 is used. The BaseAdapter class 302 implements an interface 702, called FrameworkAdapter interface 702, which functions as a superclass for all software architecture 100 adapter interfaces and facilitates adding behavior that will be common to all of them.

[0061] Next, the software architecture 100 provides a navigation adapter class 704, called the BaseNavigationAdapter class 704, which extends the BaseAdapter class 302 and functions as a superclass for all adapter classes associated with the navigation service 204. This BaseNavigationAdapter class 704 facilitates adding behavior that will be common to just the navigation service adapter objects 212.

[0062] The BaseNavigationAdapter class 704 implements an adapter interface 304 called NavigationAdapter interface 304. This is the adapter interface 304 specified by the navigation service 204. The NavigationAdapter interface 304 provides methods to be implemented by the BaseNavigationAdapter class 704 so that the navigation manager object 210 (i.e., NavigationManager object 210)

can communicate with the navigation adapter objects 212 instantiated from the BaseNavigationAdapter class 704. In one implementation, the NavigationAdapter interface 304 can add methods to process incoming calls, to determine if the cached adapter should be invalidated, and to display a page as the response to an end-user.

[0063] The software architecture 100 instantiates one or more adapter objects 212 from the BaseNavigationAdapter class 704. In one implementation, as shown in Figure 7, a NavigationAdapter object 212 is instantiated from the BaseNavigationAdapter class 704. The NavigationAdapter object 212 loads configuration data from an adapter configuration file 216, such as the NavigationAdapter configuration 216. The NavigationAdapter object 212 can use known J2EE interfaces to carry out navigation functionality, for instance, by using J2EE application component interfaces 306 such as JSPPage and Servlet.

[0064] Figure 8 is a UML diagram illustrating one implementation of a business process architecture service 206 used in the software architecture 100. Similar to the NavigationManager object 210 in Figure 7, the software architecture 100 uses the BaseManager class 300 to instantiate a manager object 210, in this case BusinessProcessManager 210, which serves as the manager object 210 for the business process service 206. The BusinessProcessManager object 210 can execute business processes by looking up the appropriate adapter object 212 to handle the business process and then delegating the call to that adapter object 212. The BusinessProcessManager object 210 loads configuration data from the manager configuration 214 (referred to here as BusinessProcessManager configuration 214).

[0065] The software architecture 100 also provides business process adapter classes for the business process service 206. Using the BaseAdapter class 302 described above, the software architecture 100 can provide a BaseBusinessProcessAdapter class 800 that extends the BaseAdapter class 302 and functions as a superclass for all of the adapter objects 212 associated with the business process service 206. The BaseBusinessProcessAdapter class 800 can

access configuration data from an adapter configuration file 216, such as the BusinessProcessAdapter configuration 216.

[0066] The BaseBusinessProcessAdapter class 800 implements an adapter interface 304, in this case BusinessProcessAdapter interface 304, as required by the business process service 206. The BusinessProcessAdapter interface 304 provides methods implemented by the business adapter objects 212 to allow the business manager object 210 (i.e., BusinessProcessManager 210) to communicate with the business adapter objects 212.

[0067] The BaseBusinessProcessAdapter class 800 also implements an application component interface 306, in this case the BusinessProcessStep interface 306. This application component interface 306 provides methods enabling the adapter objects 212 to communicate with application components 202 that contain business logic for executing business process steps. For example, an application component 202 can be included that contains business logic invoked as part of a business process. A business process is the logical grouping of a sequence of steps.

[0068] The software architecture 100 can provide a BusinessProcessAdapter class (not shown) that extends the BaseBusinessProcessAdapter class and contains most of the default behavior for the adapter objects 212 used in the business process service 206. The adapter objects 212 can then be instantiated from either the BaseBusinessProcessAdapter class 800 or the BusinessProcessAdapter class, depending on the functionality required.

[0069] Finally, because there is specialized behavior for adapter objects 212 that interface with HTTP requests (e.g., over the Internet) or with Enterprise Java Beans (EJB), separate objects can be provided for these functions. As shown in Figure 8, the software architecture 100 instantiates a WebBusinessProcessAdapter object 212 and an EJBBusinessProcessAdapter object 212. Incoming HTTP calls are handled by the WebBusinessProcessAdapter object 212, while ELB calls are handled by the EJBBusinessProcessAdapter object 212. These are just two of the many possible adapter objects 212 that can be implemented by the business process service 206.

[0070] Figure 9 is a UML diagram illustrating one implementation of a persistence architecture service 206 used in the software architecture 100. In this implementation, the software architecture 100 uses the BaseManager class 300 to instantiate a manager object 210 for the persistence service 208 referred to as the PersistenceManager object 210. The PersistenceManager object 210 can use adapter objects 212 to carry out methods for interacting with the persistence mechanism, such as a database, an extensible markup language (XML) file, or other storage devices. The PersistenceManager object 210 can also add methods to enable persistence functionality, such as a method to delete application objects, a method to call an adapter object 212, a method to store application objects, and a method to retrieve application objects. The PersistenceManager object 210 loads configuration data from the manager configuration 214, referred to as the PersistenceManager configuration 214.

[0071] The software architecture 100 provides persistence adapter classes for the persistence service 208. In this implementation, the software architecture 100 provides an adapter class called the BasePersistenceAdapter class 900 that extends the provided BaseAdapter class 302 and can be used as a superclass for the persistence adapter objects 212. A PersistenceAdapter interface 304 is provided for the BasePersistenceAdapter 900 to establish methods, implemented by the adapter objects 212, that enable communications with the manager object 210 (i.e., PersistenceManager object 210), as well as whatever persistence mechanism the software employs. The BasePersistenceAdapter 900 can also load configuration data from the PersistenceAdapter configuration 216.

[0072] Next, the software architecture 100 provides two subclasses for the BasePersistenceAdapter class 900, a BaseDelegatePA class 902 and a BaseCompositePA class 904. The BaseDelegatePA class 902 is a superclass for delegate persistence adapters that can be combined together by composite persistence adapters. The BaseCompositePA class 904 functions as a superclass for the composite persistence adapters, which can combine multiple delegate persistence adapters.

[0073] The BaseDelegatePA class 902 can have a number of subclasses that are used to instantiate adapter objects 212 for use in the persistence service 208. For instance, a DaoPA class 906 extends the BaseDelegatePA class 902 and utilizes data access objects for the persistent storage of a business object. Similarly, an EntityPA class 908 utilizes entity EJBs for the persistent storage of a business object, and a StrategyPA class 910 utilizes strategy session EJBs for the persistent storage of a business object. Application component interfaces 306 can be implemented to enable communications between adapter objects 212 instantiated from these subclasses and their associated application components 202.

[0074] The BaseCompositePA class 904 can also have a number of subclasses that are used to instantiate adapter objects 212 for the persistence service 208. For instance, the BaseCompositePA class 904 can include a DAOEntityStrategyPA class 912 to utilize data access objects, entity EJBs, and strategy session EJBs for the persistent storage of a business object, a DAOLEntityStrategyPA class (not shown) to utilize data access objects, local entity EJBs, and strategy session EJBs for the persistent storage of a business object, a DAOStrategyPA class (not shown) to utilize data access objects and strategy session EJBs for the persistent storage of a business object, an EntityStrategyPA class (not shown) to utilize both entity EJB and strategy session EJBs for the persistent storage of a business object, and a FastLaneReaderPA class (not shown) to utilize data access objects, entity EJBs, and strategy session EJBs for the persistent storage of a business object, except when handling collections. In one implementation, application components 202 can be provided to perform object relational mapping between a business object and the persistence storage. Other application components 202 can contain other logic needed for business object persistence.

[0075] The software architecture 100 can provide further manager/adapter classes and architecture services in addition to the ones described above. Figure 10 is a UML diagram of one implementation of a logging architecture service provided by the software architecture 100. Logging functionality outputs

messages from the application for many purposes including debugging, error handling, and informational purposes. The logging service can log messages of different severities to some standard repository.

[0076] As discussed above, the software application 100 provides a BaseManager class 300 and a BaseAdapter class 302 from which manager and adapter objects or classes can be generated. In the logging service, the software architecture 100 instantiates a LogManager object 210 from the BaseManager class 300. The LogManager object 210 serves as the manager object 210 of the logging service and loads configuration data from a LogManager configuration 214. The LogManager class can define methods to log a message with the debug severity and to log a message with the error severity.

[0077] The software architecture 100 also generates a BaseLogAdapter class 1000 from the BaseAdapter class 304. The BaseLogAdapter class 1000 implements a LogAdapter interface 304 and loads configuration data from a LogAdapter configuration 216. Adapter objects 212 can be instantiated from the BaseLogAdapter class 1000. For instance, a SystemOutPrintLA object 212 and a FileLA object 212 can be instantiated. The SystemOutPrintLA object 212 logs all messages to standard output and the FileLA object 212 logs all messages to a file.

[0078] Figure 11 is a UML diagram of one implementation of an application state management architecture service provided by the software architecture 100. This service is used to cache and retrieve data, for example, to manage any content that needs to be stored on the server for the convenience of retrieving it at a later time. The software application 100 again provides a BaseManager class 300 and a BaseAdapter class 302 from which manager and adapter objects or classes can be generated. In the application state management service, the software architecture 100 instantiates a ContentManager object 210 from the BaseManager class 300. The ContentManager object 210 serves as the manager object 210 of the application state management service and loads configuration data from a ContentManager configuration 214. The

ContentManager class can define methods to retrieve content and to store content.

[0079] The software architecture 100 also generates a BaseContentAdapter class 1100 from the BaseAdapter class 304. The BaseContentAdapter class 1100 implements a ContentAdapter interface 304 and loads configuration data from a ContentAdapter configuration 216. Subclasses to the BaseContentAdapter class 1100 can be provided to instantiate adapter objects 212, or adapter objects 212 can be instantiated directly from the BaseContentAdapter class 1100. In an implementation, the software architecture provides the following adapter classes or adapter objects 212: a CookieCA adapter class to deal with content stored or retrieved from cookies; an EJBSessionCA adapter class to store and retrieve content from a session EJB; a FileProxyCA adapter class to allows access to files for the business application; an HttpRequestCA adapter class to store and retrieve content from an HTTP request; an HttpSessionCA adapter class to store and retrieve content from an HTTP session (not shown); and a MapApplicationCA adapter class to store and retrieve content in a application wide repository backed by a Java Map collection object. Using the application state management service masks the intricacies of dealing with the underlying storage mechanisms.

[0080] Figure 12 is a UML diagram of one implementation of a data marshalling architecture service provided by the software architecture 100. Data marshalling handles the conversion of data from one format to another. Each adapter object 212 in the data marshalling service can have a different mechanism for converting data, and can support one or more types of conversions. A typical example of a conversion is converting data from objects to XML.

[0081] As before, the software application 100 provides a BaseManager class 300 and a BaseAdapter class 302 from which manager and adapter objects or classes can be generated. In the data marshalling service, the software architecture 100 instantiates a MappingManager object 210 from the BaseManager class 300. The MappingManager object 210 serves as the

manager object 210 of the data marshalling service and loads configuration data from a MappingManager configuration 214. The MappingManager class can define methods such as for extracting information contained in an object and for populating an object.

[0082] The software architecture 100 also generates a BaseMappingAdapter class 1200 from the BaseAdapter class 304. The BaseMappingAdapter class 1200 implements a MappingAdapter interface 304 and loads configuration data from a MappingAdapter configuration 216. The software architecture 100 can instantiate an adapter object 212 from the BaseMappingAdapter class 1200, such as a MappingAdapter object 212. The MappingAdapter object 212 can perform XML data binding for both marshaling and unmarshaling, for instance, marshaling objects into their XML representation and unmarshaling XML representations to generate and/or populate objects. The MappingAdapter 212 loads configuration data from a MappingAdapter configuration 216 and communicates with application components 202 through a Mapper application component interface 306.

[0083] In an implementation, the software architecture 100 can provide one adapter class for handling the marshaling and unmarshaling of business objects and business object collections, one adapter class for handling the marshaling and unmarshaling of event objects containing business objects, business object collections, application exceptions, and process names, one adapter class for handling the marshaling and unmarshaling of java.util.Map objects, one adapter class to provide a default adapter implementation for the MappingManager and to function as a superclass for custom adapter objects 212 associated with the MappingManager, one adapter class to use session or entity EJBs for the generation and obtaining of primary keys for business objects, and one adapter class to use session EJBs for the generation and obtaining of primary keys for business objects.

[0084] Figure 13 is a UML diagram of one implementation of a key management architecture service provided by the software architecture 100. Key management handles the generation of unique identifiers (i.e., keys) for relevant

objects. Each adapter object 212 used in this service can use a different mechanism to generate unique keys. The software application 100 again provides a BaseManager class 300 and a BaseAdapter class 302 from which manager and adapter objects or classes can be generated. In the key management service, the software architecture 100 instantiates a KeyManager object 210 from the BaseManager class 300. The KeyManager object 210 serves as the manager object 210 of the key management service and loads configuration data from a KeyManager configuration 214. The KeyManager class can define methods to construct a new primary key of the correct type for the object with a valid value and to return the field name that is used by the software application to contain the primary key value when marshaling and unmarshaling data.

[0085] The software architecture 100 also generates a BaseKeyAdapter class 1300 from the BaseAdapter class 304. The BaseKeyAdapter class 1300 implements a KeyAdapter interface 304 and loads configuration data from a KeyAdapter configuration 216. The software architecture 100 can instantiate an adapter object 212 from the BaseKeyAdapter class 1300, such as an EntitySessionKA 212 that uses either session or entity EJBs for the creation and obtaining of Primary Keys for business objects, a HLStringKA 212 that uses session EJBs for the creation and obtaining of primary keys for business objects, and a HLIntegerKA 212 that also uses session EJBs for the creation and obtaining of Primary Keys for business objects.

[0086] Figure 14 is a flowchart illustrating one implementation of how the client retrieves business functionality from the software architecture. The client begins by calling a manager object in the software architecture (step 1400). If no manager object exists at the moment the business application framework calls into the software architecture, a manager object can be instantiated from the appropriate manager class.

[0087] Once the manager object is instantiated, the manager object loads and caches the manager configuration data instructing the manager object on how to delegate different calls from the business application framework to the

appropriate adapter object (step 1402). The manager object parses the call and selects an adapter class based at least in part on the instructions in the manager configuration, and the manager object instantiates and caches an adapter object from the selected adapter class (step 1404). The manager object calls the instantiated adapted object and delegates to it the call from the client (step 1406).

[0088] The adapter object, after it has been instantiated, loads and caches the adapter configuration data instructing the adapter object on how to handle different calls from the client (step 1408). The adapter object instantiates and caches the appropriate application component or components to handle the request from the client based on the data in the adapter configuration (step 1410). The application components can be provided as classes by the user; application components can then be instantiated from the appropriate class when needed. Alternatively, the application components can be provided as objects and do not have to be instantiated by the adapter object.

[0089] The adapter object can execute any technical code that is required by the request from the client (step 1412). The adapter object then calls the application component (step 1414), and the application component executes the business code for which it was developed (step 1416). The execution of the business code generally results in some form of output being returned to the client.

[0090] Figure 15 is a flowchart describing one implementation of the operation of the navigation service 204. The navigation service displays the graphical user interface (GUI) to an end-user and controls the user interface flow, known as navigation. In this implementation, the navigation service instantiates a NavigationManager object as described above, and implements the NavigationAdapter interface. The NavigationManager object can instantiate an adapter object such as a NavigationAdapter object. Application components, such as application components used for the user interface (e.g., application components to generate HTML pages and JSP pages), can be instantiated by the adapter object.

[0091] When a request from an end-user is received from the client, it is sent as a call into the navigation service and received by the NavigationManager object (step 1500). The NavigationManager object loads and reads the navigation configuration (step 1502), and based at least in part on the instructions in the configuration, selects the NavigationAdapter to handle the call from the client (step 1504).

[0092] The NavigationManager object loads and caches the selected NavigationAdapter object (step 1506), and passes the call to the NavigationAdapter object. The NavigationAdapter object then loads and reads the navigation configuration (step 1508). In some implementations, the configuration for the manager is different than the configuration for the adapters, while in other implementations the same configuration can be used for both. Next, the NavigationAdapter object, based at least in part on the instructions in the configuration, determines which business process is being requested by the call (step 1510).

[0093] To execute the business process, the NavigationAdapter object calls the BusinessProcessManager from the business process service and passes the call from the client on to the BusinessProcessManager object (step 1512). The call from the client is a request that generally contains the name of the business process to execute and the user data necessary to carry out the process.

[0094] After the BusinessProcessManager object processes the call, it returns data to the NavigationAdapter object (step 1514). The NavigationAdapter object reads the navigation configuration that contains rules specifying which user interface component to display based on the data returned from the BusinessProcessManager object (step 1516). Based on this data, the NavigationManager object displays the appropriate user interface component to the client (step 1516).

[0095] Figure 16 is a flowchart describing one implementation of the operation of the business process service 206. As explained above, the business process service manages and executes the business logic of the application. In this implementation, the business process service can instantiate and use a

BusinessProcessManager object, and can implement a BusinessProcessAdapter interface as its adapter interface. The business process service can instantiate adapter objects as required, the EJBBusinessProcessAdapter and the WebBusinessProcessAdapter being two examples. The business logic requirements map to business process steps which implement the business logic and are executed by the BusinessProcessManager object.

[0096] The BusinessProcessManager is generally called from another software component, such as the navigation service, to execute a named business process (step 1600). For example, if the business process is for banking transactions, the named business process can be called TransferFundsProcess. The incoming call generally includes the name of the business process to execute and the data required to execute the process. The BusinessProcessManager object reads the business process service configuration (step 1602) and selects the appropriate adapter object (e.g., the EJBBusinessProcessAdapter object) specified for the business process requested in the call (step 1604). The configuration provides instructions used by the BusinessProcessManager object to select and instantiate the appropriate adapter object.

[0097] The BusinessProcessManager loads and instances the selected adapter object (step 1606). The BusinessProcessManager then calls that adapter object and passes on the name of the requested business process and the data required to execute the process to the adapter object (step 1608). The adapter object reads the business service configuration and finds the definition for the requested business process (step 1610). The configuration also enables the adapter object to locate and retrieve the appropriate application components specified for the named business process (step 1610). The adapter object then loads and instances the appropriate application components needed to carry out the requested business process (step 1612). For example, if the requested business process is the TransferFundsProcess mentioned above, the appropriate application components could be a DebitAccountStep object and a CreditAccountStep object.

[0098] The adapter object calls each application component sequentially passing the data (step 1614), and the application components execute the requested business functionality (step 1616). For example, in the TransferFundsProcess, the DebitAccountStep object can remove money from a first account, and the CreditAccountStep object can add money to a second account.

[0099] Figure 17 is a flowchart describing one implementation of the operation of the persistence service 208. The persistence service performs persistence operations (i.e., create, retrieve, update, and delete) on the objects/data of the business application. Different persistence adapters are provided for different technical persistence mechanisms. In this implementation, the persistence service can instantiate and use a PersistenceManager object, and can implement a PersistenceAdapter interface. The persistence service can also instantiate adapter objects, such as a DaoPA object, an EntityPA object, a StrategyPA object, an EmailPA object, a FastLaneReaderPA object, as well as many other adapter objects. The application components that can be called by these adapter objects can include a Dao object, a StrategyEJB object, and an EntityEJB object.

[0100] In the persistence service, the PersistenceManager is generally called by a software component, such as a business application component from the business process service (step 1700). The call includes the appropriate data needed to carry out the persistence functionality. The PersistenceManager object reads persistence service configuration (step 1702) and selects the appropriate persistence adapter object based on the data that is passed in and based on the instructions in the configuration (step 1704). The PersistenceManager object then loads and instances the selected adapter object (e.g., the DaoPA object) (step 1706), and passes the necessary data on to the adapter object (step 1706).

[0101] The persistence adapter object loads and reads the persistence configuration (step 1708) and selects the appropriate application component to handle the requested persistence functionality (step 1710). The persistence

adapter object then instances and calls the application component and passes the data on to the application component (step 1712). The return data from the application component after the functionality is carried out is passed back up to the caller of the PersistenceManager (step 1714).

[0102] The invention can be implemented in digital electronic circuitry, or in computer hardware, firmware, software, or in combinations of them. The invention can be implemented as a computer program product, i.e., a computer program tangibly embodied in an information carrier, e.g., in a machine readable storage device or in a propagated signal, for execution by, or to control the operation of, data processing apparatus, e.g., a programmable processor, a computer, or multiple computers. A computer program can be written in any form of programming language, including compiled or interpreted languages, and it can be deployed in any form, including as a stand alone program or as a module, component, subroutine, or other unit suitable for use in a computing environment. A computer program can be deployed to be executed on one computer or on multiple computers at one site or distributed across multiple sites and interconnected by a communication network.

[0103] Method steps of the invention can be performed by one or more programmable processors executing a computer program to perform functions of the invention by operating on input data and generating output. Method steps can also be performed by, and apparatus of the invention can be implemented as, special purpose logic circuitry, e.g., an FPGA (field programmable gate array) or an ASIC (application specific integrated circuit).

[0104] Processors suitable for the execution of a computer program include, by way of example, both general and special purpose microprocessors, and any one or more processors of any kind of digital computer. Generally, a processor will receive instructions and data from a read only memory or a random access memory or both. The essential elements of a computer are a processor for executing instructions and one or more memory devices for storing instructions and data. Generally, a computer will also include, or be operatively coupled to receive data from or transfer data to, or both, one or more mass storage devices

for storing data, e.g., magnetic, magneto optical disks, or optical disks. Information carriers suitable for embodying computer program instructions and data include all forms of non volatile memory, including by way of example semiconductor memory devices, e.g., EPROM, EEPROM, and flash memory devices; magnetic disks, e.g., internal hard disks or removable disks; magneto optical disks; and CD ROM and DVD-ROM disks. The processor and the memory can be supplemented by, or incorporated in special purpose logic circuitry.

[0105] The software architecture of the invention therefore separates the difficult technical challenges associated with the production of a business application. As described herein, the software architecture accomplishes this by defining how to develop business components in a way that is independent of the underlying technologies. Developers who use the software architecture of the invention can invest significantly less time becoming familiar with distributed infrastructure solutions, and can spend more time concentrating on implementing their application-specific business logic. In such a case, these developers need only to understand the business-level components.

[0106] The invention has been described with reference to specific implementations. Other implementations of the invention will be apparent to those of ordinary skill in the art. For example, the functionality of the software architecture 100 can be used for any type of software application, not just business applications. It is, therefore, intended that the scope of the invention not be limited to the implementations described above.